# OpenUH Compiler: Exploring Language Constructs and Their Implementation

Barbara Chapman

University of Houston

April, 2007

# Agenda

- OpenUH compiler
- OpenMP language extensions
- Compiler – Tools interactions
- Compiler cost modeling

# The Open64 Compiler Suite

An optimizing compiler suite for C/C++ and Fortran77/90 on Linux/IA-64 systems

- Open-sourced by SGI from Pro64 compiler
- State-of-the-art intra- & interprocedural analysis and optimizations
- 5 levels of uniform IR (WHIRL), with IR-to-source "translators": whirl2c & whirl2f
- Used for research and commercial purposes: Intel, HP, QLogic, STMicroelectronics, UPC, CAF, U Delaware, Tsinghua, Minnesota …

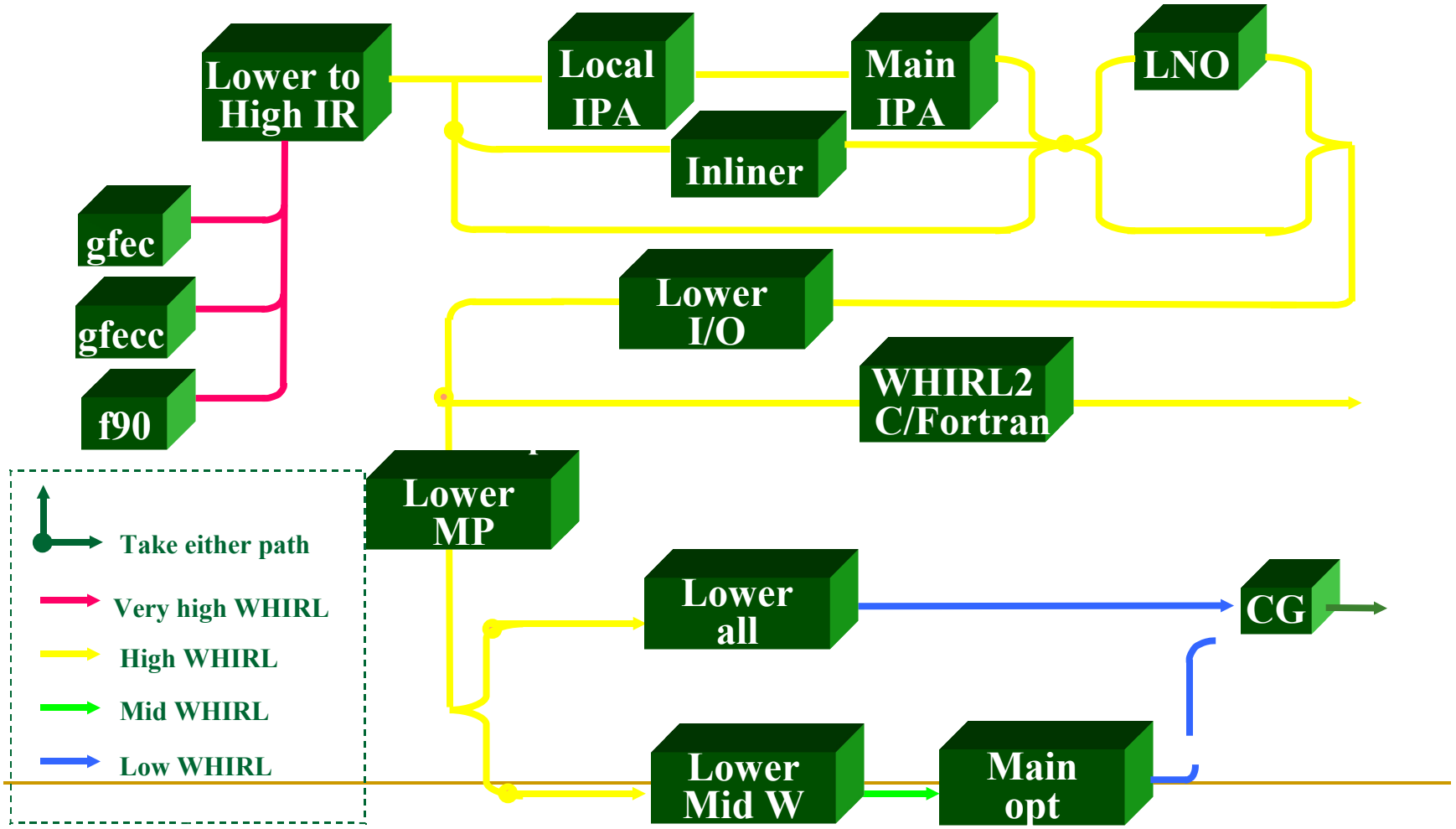
# OpenUH: A Reference OpenMP Compiler

- Based on Open64
  - Integrates features from other major branches: Pathscale, ORC, UPC,…
- Complete support for OpenMP 2.5 in C/C++ and Fortran
- Modularized and complete optimization framework
- Stable, portable
- Freely available and open source
- Available on most Linux/Unix  platforms

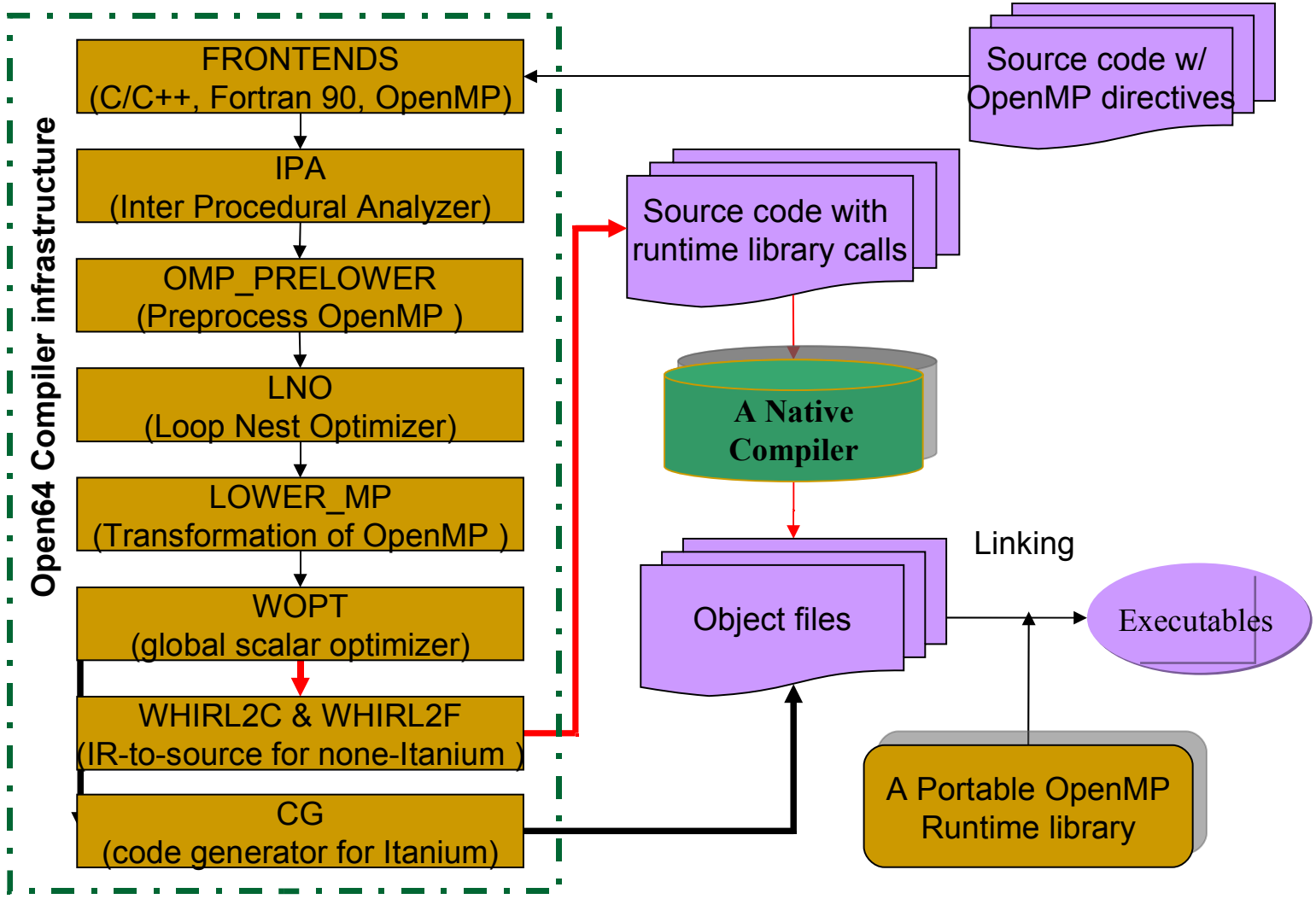
# OpenUH: A Reference OpenMP Compiler

- Facilitates research and development
  - Testbed for new language features
  - New compiler transformations
  - Interactions with variety of programming tools
- Currently installed for experimentation at Cobalt@NCSA and Columbia@NASA
  - Cobalt: 2x512 processors, Columbia: 20x512 processors

# Major Modules in Open64



# OpenUH Compiler Infrastructure



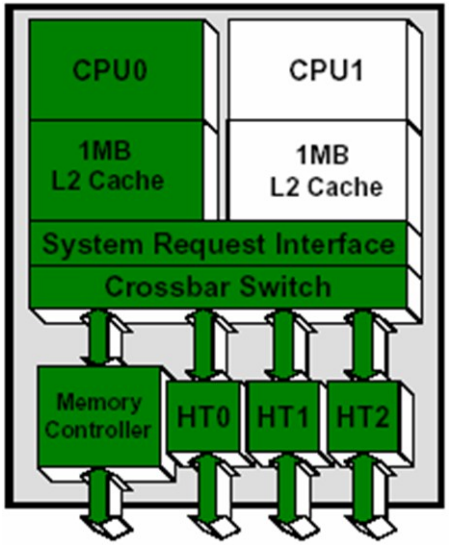
# OpenMP Implementation in OpenUH

- Frontends:
  - Parse OpenMP pragmas
- OMP_PRELOWER:
  - Preprocessing
  - Semantic checking
- LOWER_MP
  - Generation of microtasks for parallel regions
  - Insertion of runtime calls
  - Variable handling, …
- Runtime library
  - Support for thread manipulations
  - Implements user level routines
  - Monitoring environment

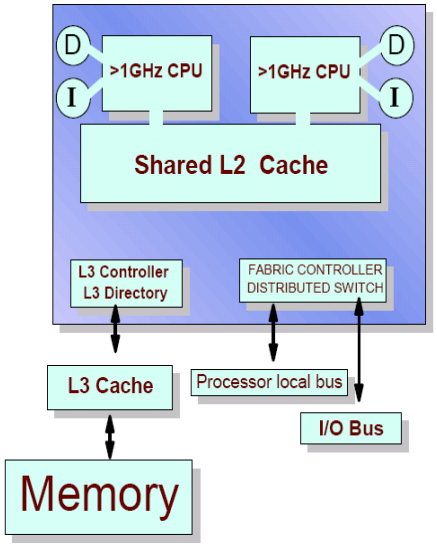
| OpenMP Code | Translation |
|---|---|
| int main(void)<br>{<br>int a,b,c;<br>#pragma omp parallel \<br>private(c)<br>do_sth(a,b,c);<br>return 0;<br>} | _INT32 main()<br>{<br>int a,b,c;<br>/* microtask */<br>void __ompregion_main1()<br>{<br>_INT32 __mplocal_c;<br>/*shared variables are kept intact,<br>substitute accesses to private<br>variable*/<br>do_sth(a, b, __mplocal_c);<br>}<br>/*OpenMP runtime calls */<br>__ompc_fork(&__ompregion_main1<br>);<br>…<br>} |

Collaboration with Tsinghua University who created OpenMP runtime

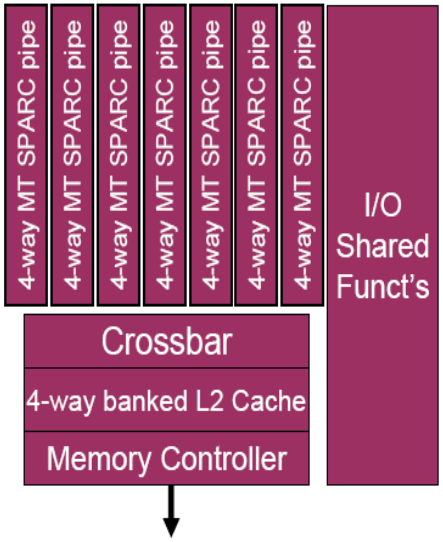
# Multicore Complexity



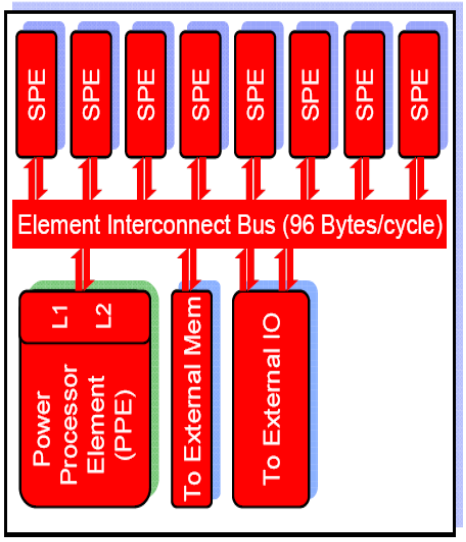

AMD dual-core




IBM Power4




Sun T-1 (Niagara)




Cell processor

- Resources (L2 cache, memory bandwidth): shared or separate
- Each core: single thread or multithreaded, complex or simplified
- Individual cores: symmetric or asymmetric (heterogeneous)

# Challenges Posed By New Architectures

We may want sibling threads to share in a workload on a multicore. But we may want SMT threads to do different things

- Hierarchical and hybrid parallelism
  - Clusters, SMPs, CMP (multicores), SMT (simultaneous multithreading), …
- Diversity in kind and extent of resource sharing, potential for thread contention
  - ALU/FP units, cache, MCU, data-path, memory bandwidth
- Homogeneous or heterogeneous
- Deeper memory hierarchy
- Size and scale

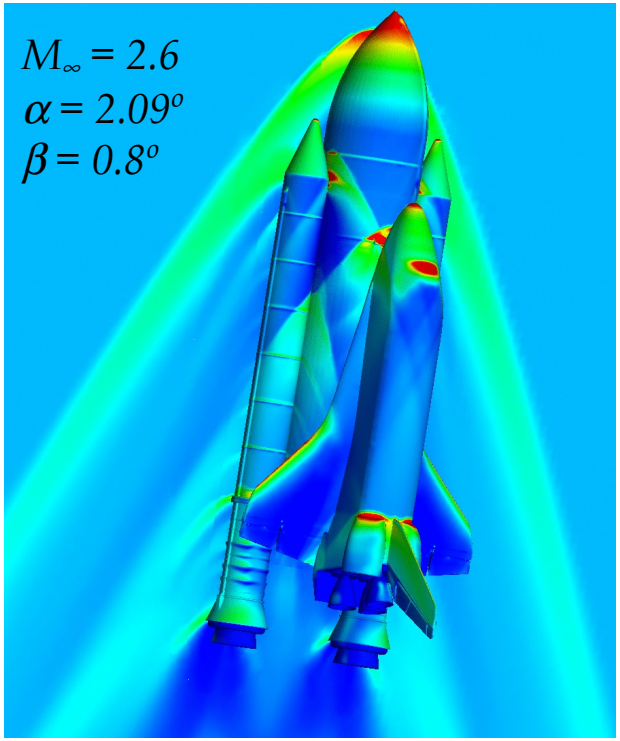
Do we want codes with multiple levels of parallelism?

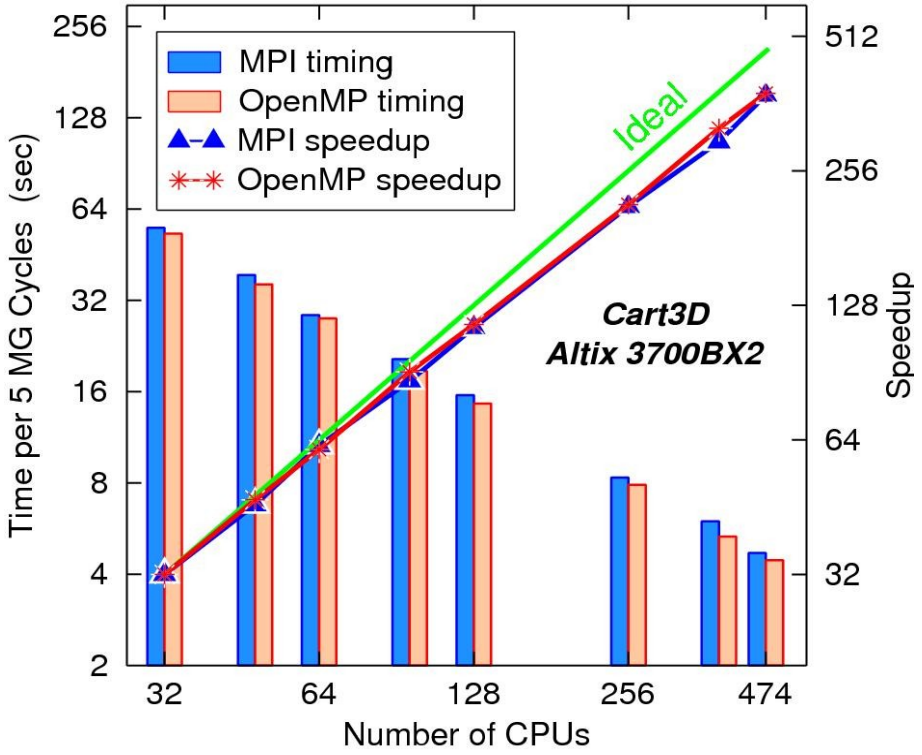
# Is OpenMP Ready for Multicore?

- Existing API
  - Designed for medium-scale SMPs: <100 threads
  - Uniform memory access
    - Hard work needed to get better data locality
- Potential difficulties using OpenMP on these platforms
  - Determining the optimal number of threads
  - Binding threads to right processor cores
  - Finding good scheduling policy and chunk size

# Cart3D OpenMP Scaling

4.7 M cell mesh Space Shuttle Launch Vehicle example




- OpenMP version uses same domain decomposition strategy as MPI for data locality, avoiding false sharing and fine-grained remote data access

OpenMP version slightly outperforms MPI version on SGI Altix 3700BX2, both close to linear scaling.

# Case Study: A Seismic Code

```
for (i=0;i<N;i++) {
    ReadFromFile(i,...);
    for (j=0; j<ProcessingNum; j++)
     for(k=0;k<M;k++){
        process_data();
      //involves several different seismic functions
   }
     WriteResultsToFile(i);
 }
```

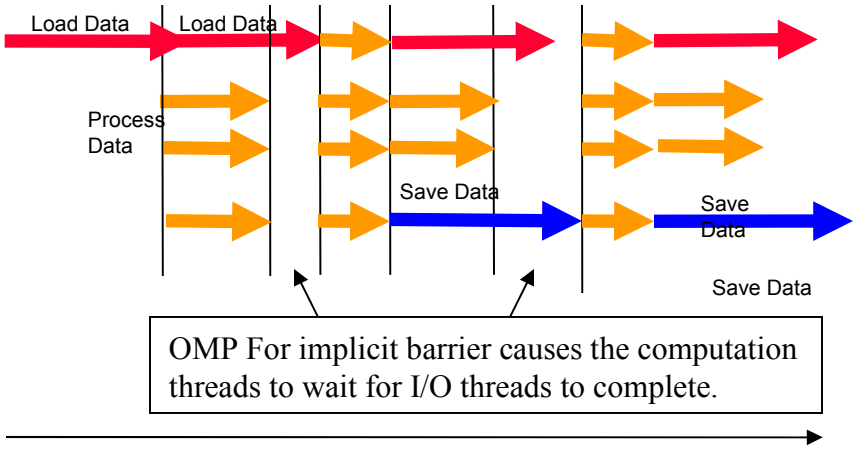
This loop is parallel

**Kingdom Suite from Seismic Micro Technology**

Goal: create OpenMP code for SMP with simultaneous multithreading (hyperthreading) enabled

# Parallel Seismic Kernel V1

```
for( j=0;  j< ProcessingNum; j++) {
  #pragma omp for schedule(dynamic)
  for(k=0; k<M; k++) {
    processing();  //user configurable functions
  }    // here is the barrier
 } end of j-loop
```




Load Data:

Process Data:

Save Data:

Idea: We want to overlap I/O with computation so that threads sharing processor (core) have different workloads

# Subteams of Threads

```
for (j=0; j<ProcessingNum;j++) {
   #pragma omp for on threads (2: omp_get_num_threads()-1 )
    for k=0; k<M;k++) {        //on threads in subteam
         ...   processing();
    }                    // barrier involves subteam only
```

- We need to avoid the global barrier early on in the loop's execution
- One way to do this would be to restrict loop execution to a subset of the team of executing threads

Idea: We want to overlap I/O with computation so that threads sharing processor (core) have different workloads

The reality: It didn't work. The barrier was global.

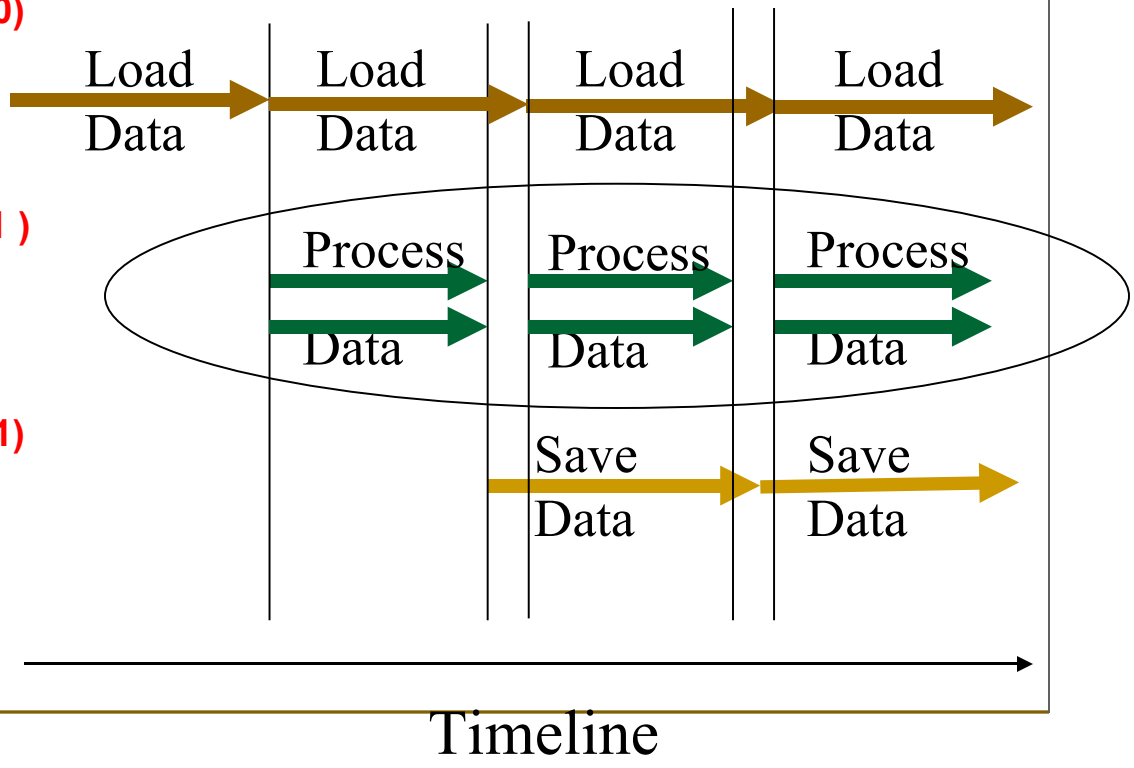
# Subteams of Threads

```
for (j=0; j<ProcessingNum;j++) {
   #pragma omp for on threads (2: omp_get_num_threads()-1 )
    for k=0; k<M;k++) {        //on threads in subteam
         ...   processing();
    }                    // barrier involves subteam only
```

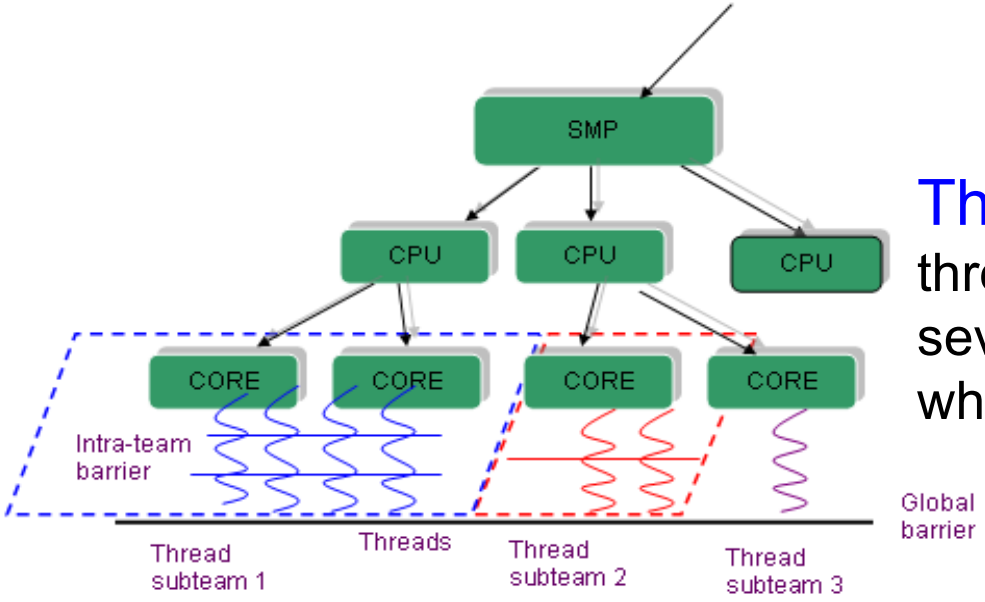
- MPI provides for definition of groups of pre-existing processes
- Why not allow worksharing among groups (or subteams) of pre-existing threads?
- Logical machine description, mapping of threads to it
  - Or simple “spread” or “keep together” notations

# Parallel Seismic Code V2

```
 Loadline(nStartLine,...);  // preload the first line of data
#pragma omp parallel
{
   for (int iLineIndex=nStartLine; iLineIndex <= nEndLine; iLineIndex++)
   {
#pragma omp single nowait          onthread(0)
  {// loading the next line data, NO WAIT!
        Loadline(iLineIndex+1,...);
   }
      for(j=0;j<iNumTraces;j++)
#pragma omp for schedule(dynamic)
onthread(2: omp_get_num_threads()-1 )

     for(k=0;k<iNumSamples;k++)
          processing();

#pragma omp barrier
#pragma omp single nowait          onthread(1)
    {
         SaveLine(iLineIndex);
     }
   }
}
```



# OpenMP Scalability: Thread Subteam




Thread Subteam: original thread team is divided into several subteams, each of which can work simultaneously.

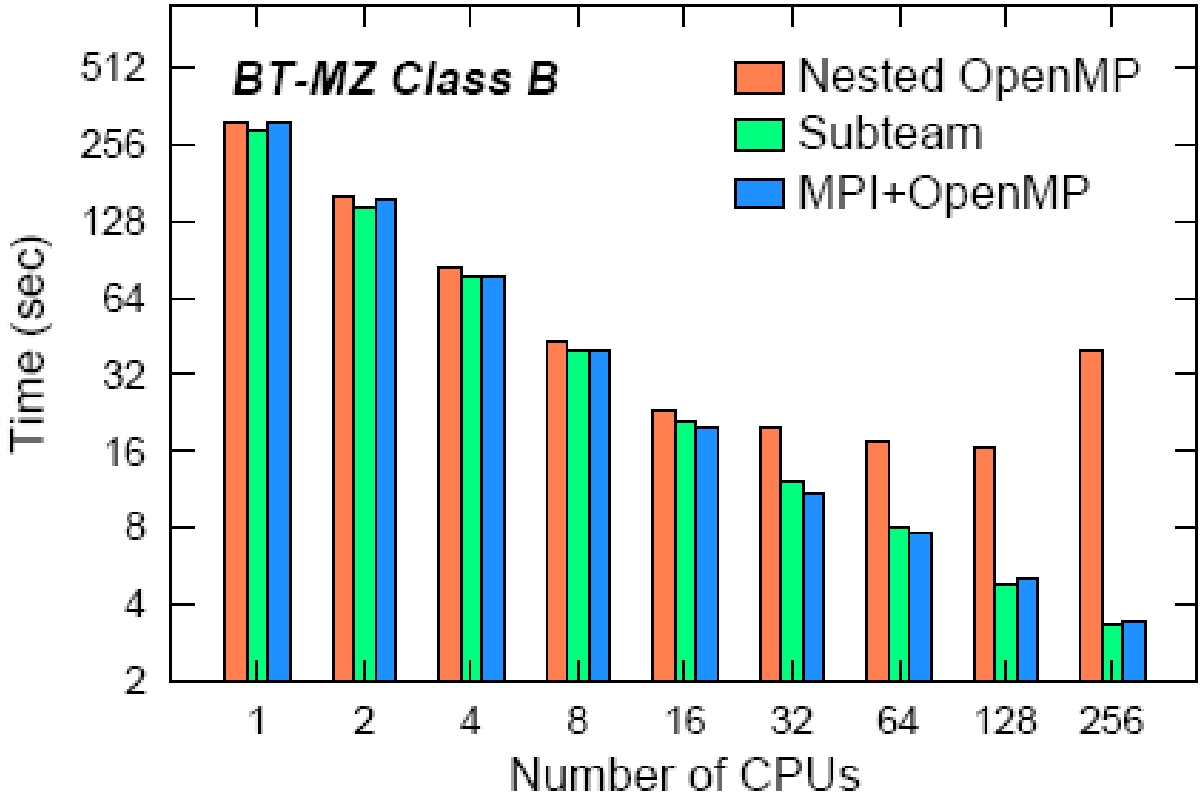
- Advantages
  - Flexible worksharing/synchronization extension
  - Low overhead because of static partition
  - Facilitates thread-core mapping for better data locality and less resource contention

# Implementation in OpenUH

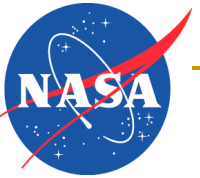
```
void * threadsubteam;
 __ompv_gtid_s1 = __ompc_get_local_thread_num();
__ompc_subteam_create(&idSet1,&threadsubteam);

/*threads not in the subteam skip the later work*/
if (!__ompc_is_in_idset(__ompv_gtid_s1,&idSet1)) goto L111;
  __ompc_static_init_4(__ompv_gtid_s1, …&__do_stride, 1, 1, &threadsubteam);
for(__mplocal_i = __do_lower; __mplocal_i <= __do_upper; __mplocal_i = __mplocal_i + 1)
 {
       .....
 }                                                          //omp for
  __ompc_barrier(&threadsubteam); /*barrier at subteam only*/

L111: /* Insert a label as the boundary between two worksharing bodies*/
 __ompv_gtid_s1 = __ompc_get_local_thread_num();
 mpsp_status = __ompc_single(__ompv_gtid_s1);
 if(mpsp_status == 1)
 {    j = omp_get_thread_num();
      printf("I am the one: %d\n", j);          //omp single
 }
 __ompc_end_single(__ompv_gtid_s1);
 __ompc_barrier (NULL); /*barrier at  the default team*/
```

- Tree-structured team and subteams in runtime library
- Threads not in a subteam skip the work in compiler translation
- Global thread IDs are converted into local IDs for loop scheduling
- Implicit barriers only affect threads in a subteam

# BT-MZ Performance with Subteams




Platform: Columbia@NASA



# Beyond OpenMP 3.0

Major thrust for 3.0 spec. supports non-traditional loop parallelism

Ideas on support for multicore / higher levels scalability

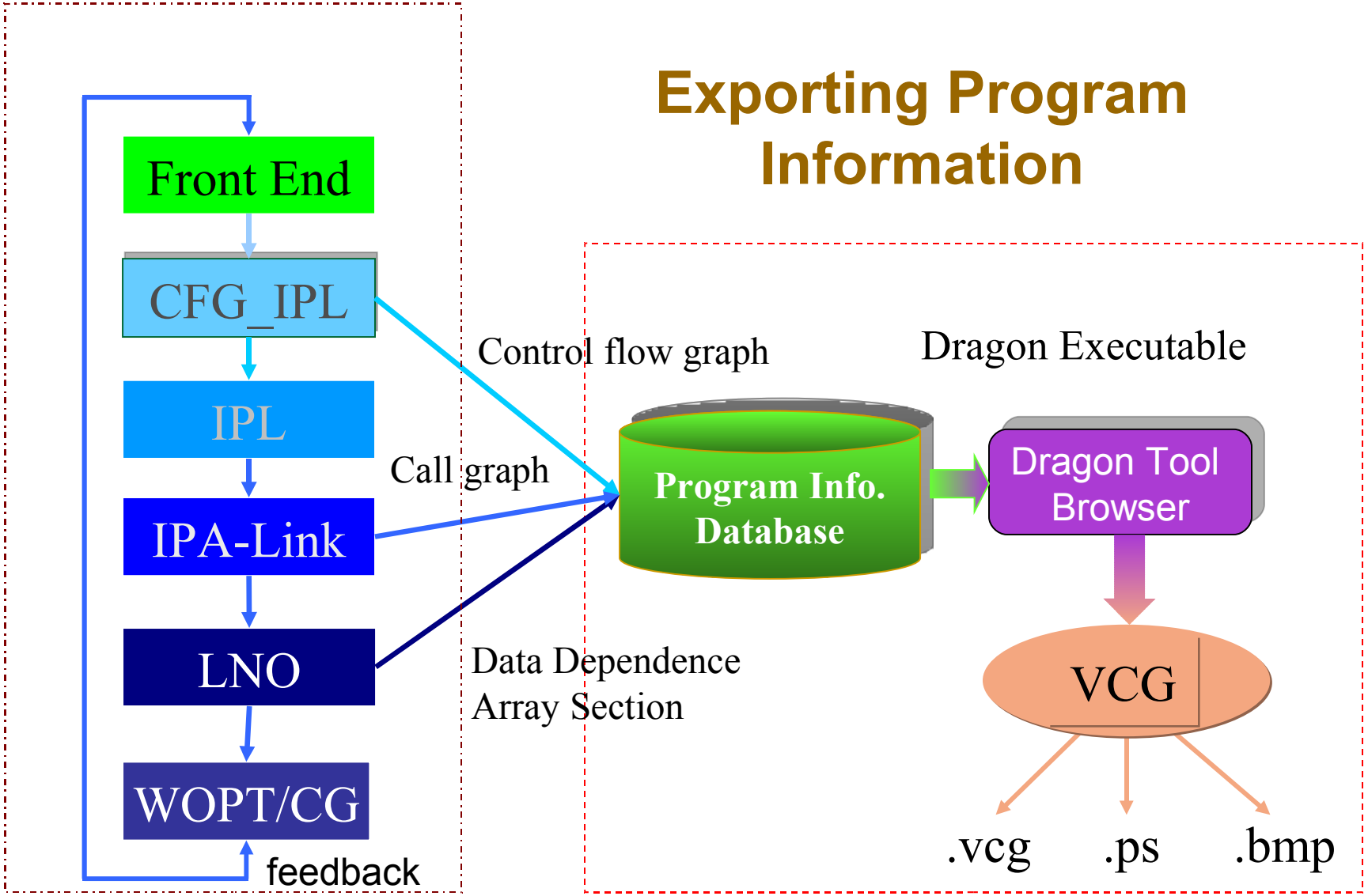
- Extend nested parallelism by binding threads in advance
  - High overhead of dynamic thread creation/cancellation
  - Poor data locality between parallel regions executed by different threads without binding
- Describe structure of threads used in computation
  - Map to logical machine, or group
- Explicit data migration
- Subteams of threads
- Control over the default behavior of idle threads

# What About The Tools?

- Typically hard work to use, steep learning curve

- Low-level interaction with user
- Tuning may be fragmentary effort
- May require multiple tools
  - Often not integrated with each other
  - Let alone with compiler
- Can we improve tools' results, reduce user effort and help compiler if they interact?

# Exporting Program Information

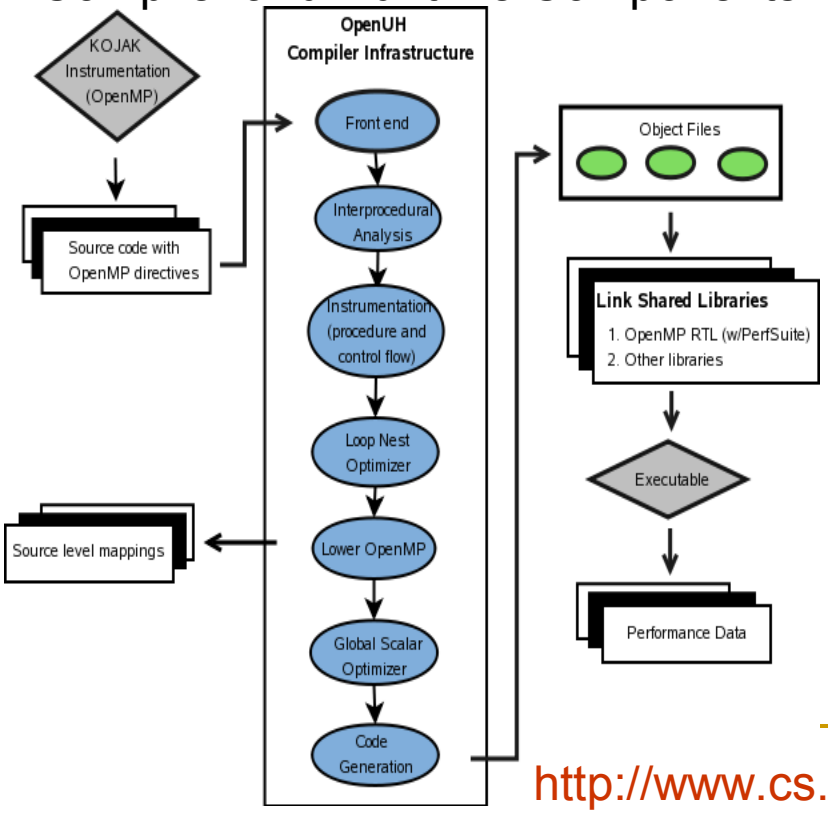



Static and dynamic program information is exported

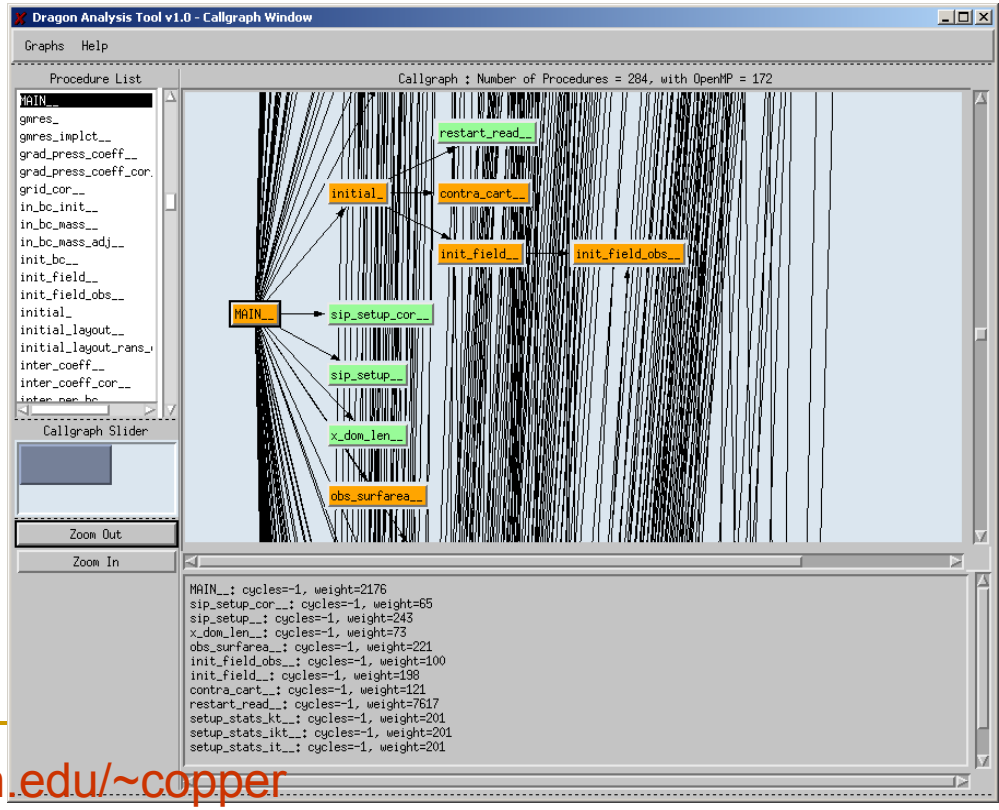
# OpenUH Tuning Environment

- Manual or automatic selective instrumentation, possibly in iterative process
- Instrumented OpenMP runtime can monitor parallel regions at low cost
- KOJAK able to look for performance problems in output and present to user

## Compiler and Runtime Components




## Selective Instrumentation analysis

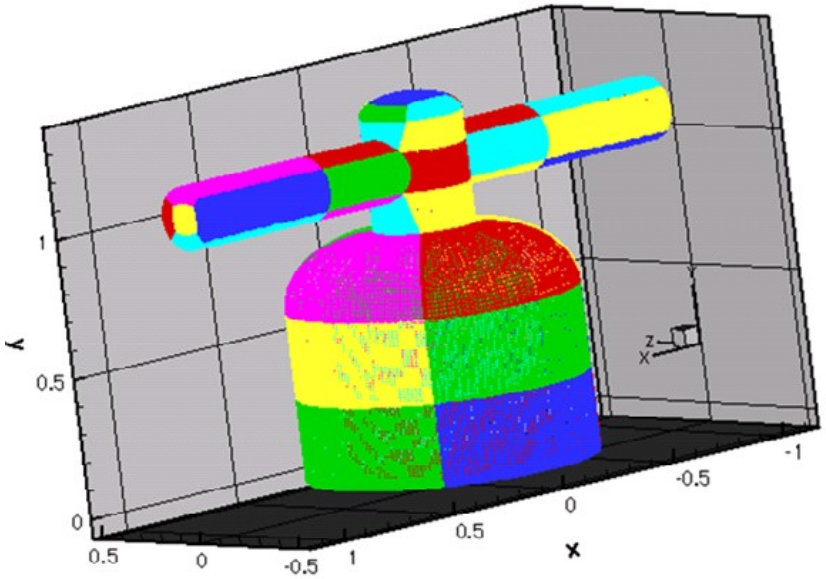



http://www.cs.uh.edu/~copper

NSF CCF-0444468

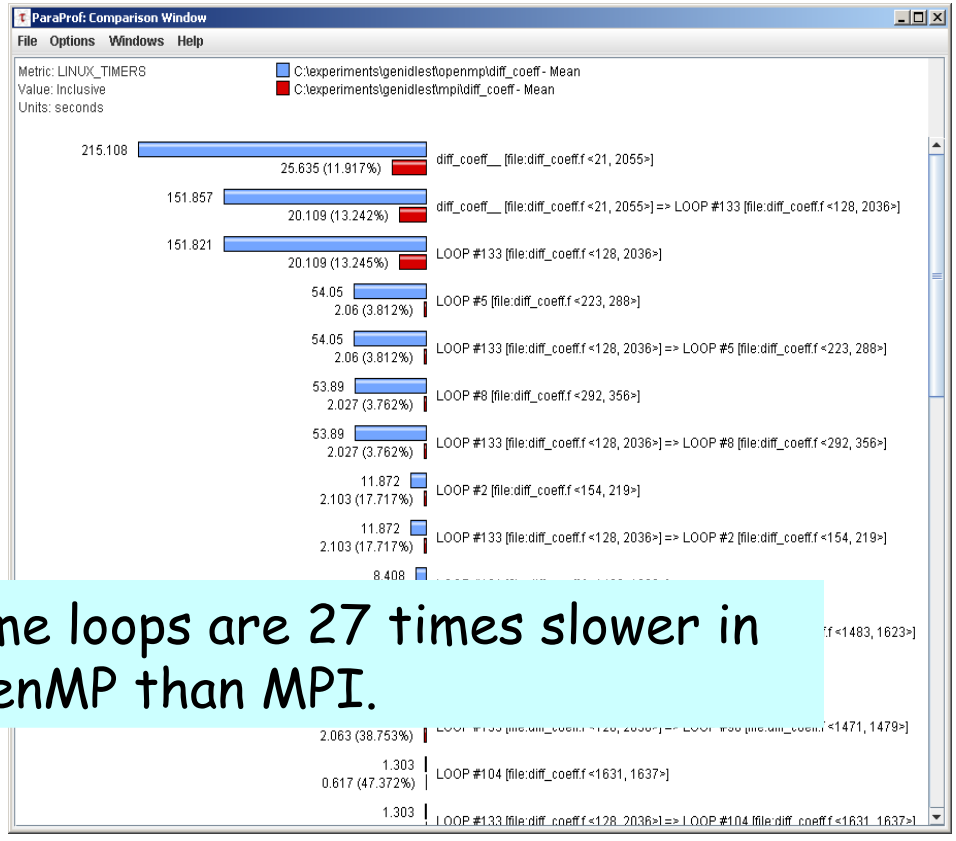
# A Performance Problem: Specification

- GenIDLEST
  - Real world scientific simulation code
  - Solves incompressible Navier Stokes and energy equations
  - MPI and OpenMP versions
- Platform
  - SGI Altix 3700
  - Two distributed shared memory systems
    - Each system with 512 Intel Itanium 2 Processors
- Thread count in e.g.: 8
- The problem: OpenMP version is slower than MPI



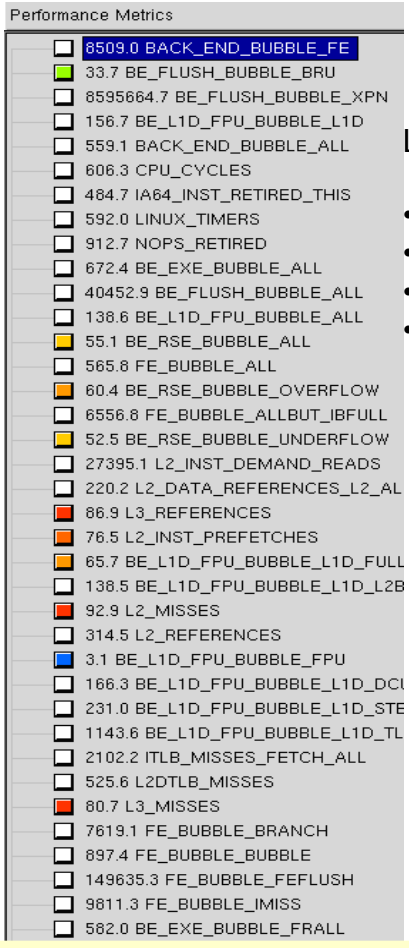
# Performance Analysis

## Procedure Timings




Some loops are 27 times slower in OpenMP than MPI.

When comparing the metrics between OpenMP and MPI using KOJAK performance algebra.




We find:

Large # of:

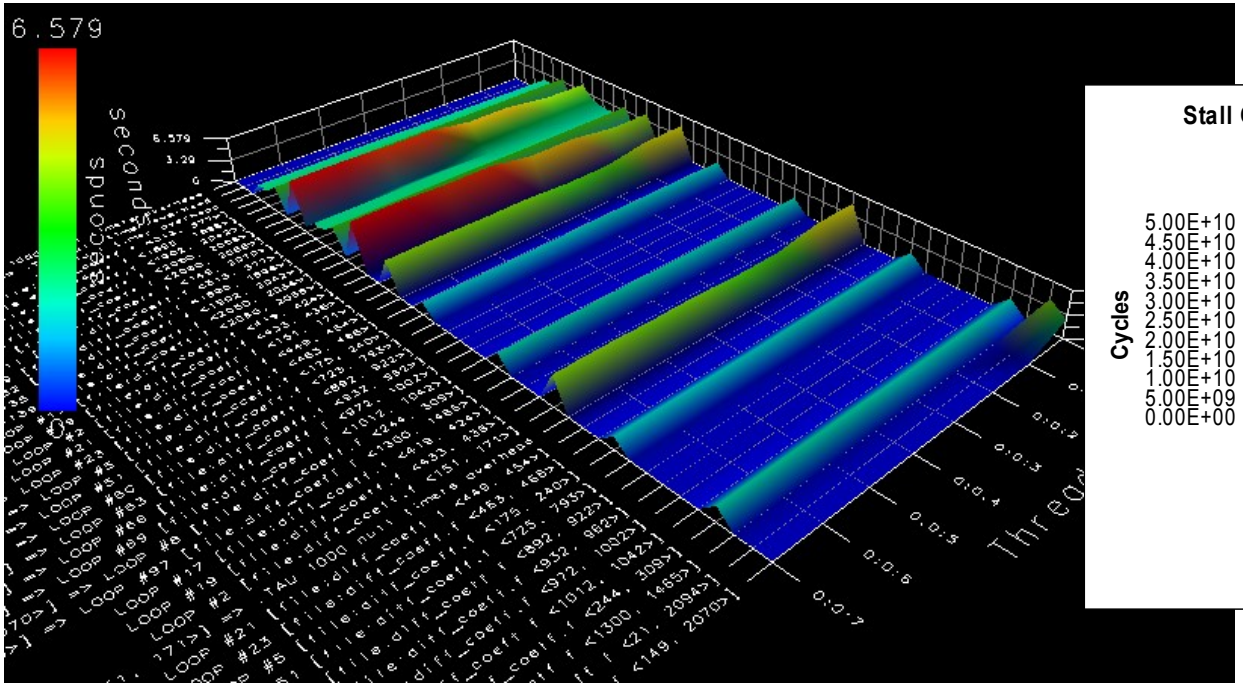
- Exceptions
- Flushes
- Cache Misses
- Pipeline stalls

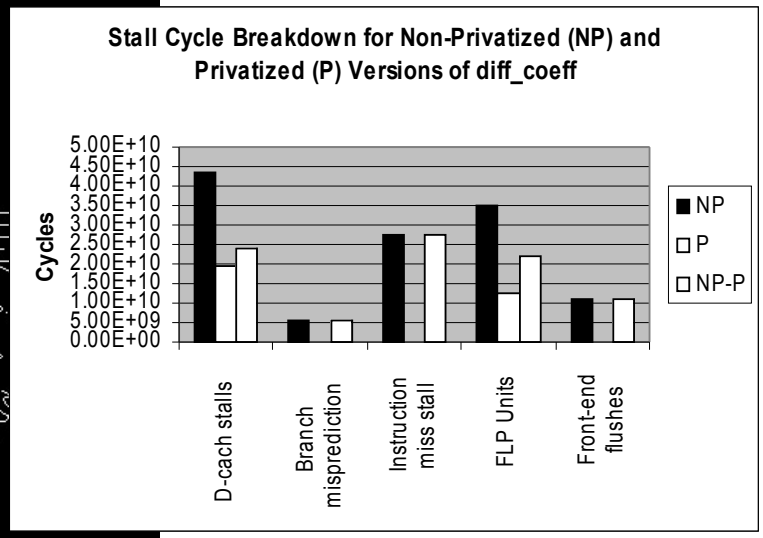
- Lower and upper bounds of thread-local computational loops are shared, and stored in same memory page and cache line
- Delays in remote memory accesses are probable causes of exceptions causing processor flushes

# Solution: Privatization

- Privatizing the arrays improved the performance of the whole program by 30% and resulted in a speedup of 10 for the problem procedure.
- Now this procedure only takes 5% of total time
- Processor Stalls are reduced significantly

OpenMP Privatized Version



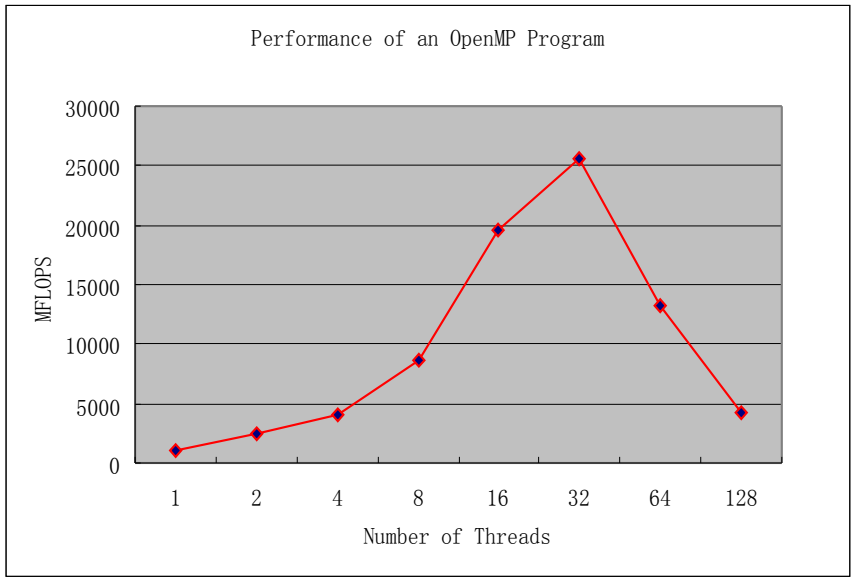



# OpenMP Platform-Awareness: Cost modeling

- Cost modeling:
  - To estimate the cost, mostly the time, of executing a program (or a portion of it) on a given system (or a component of it) using compilers, runtime systems, performance tools, etc.
- An OpenMP cost model is critical for:
  - OpenMP compiler optimizations
  - Adaptive OpenMP runtime support
  - Load balancing in hybrid MPI/OpenMP
  - Targeting OpenMP to new architectures: multicore
    - Complementing empirical search

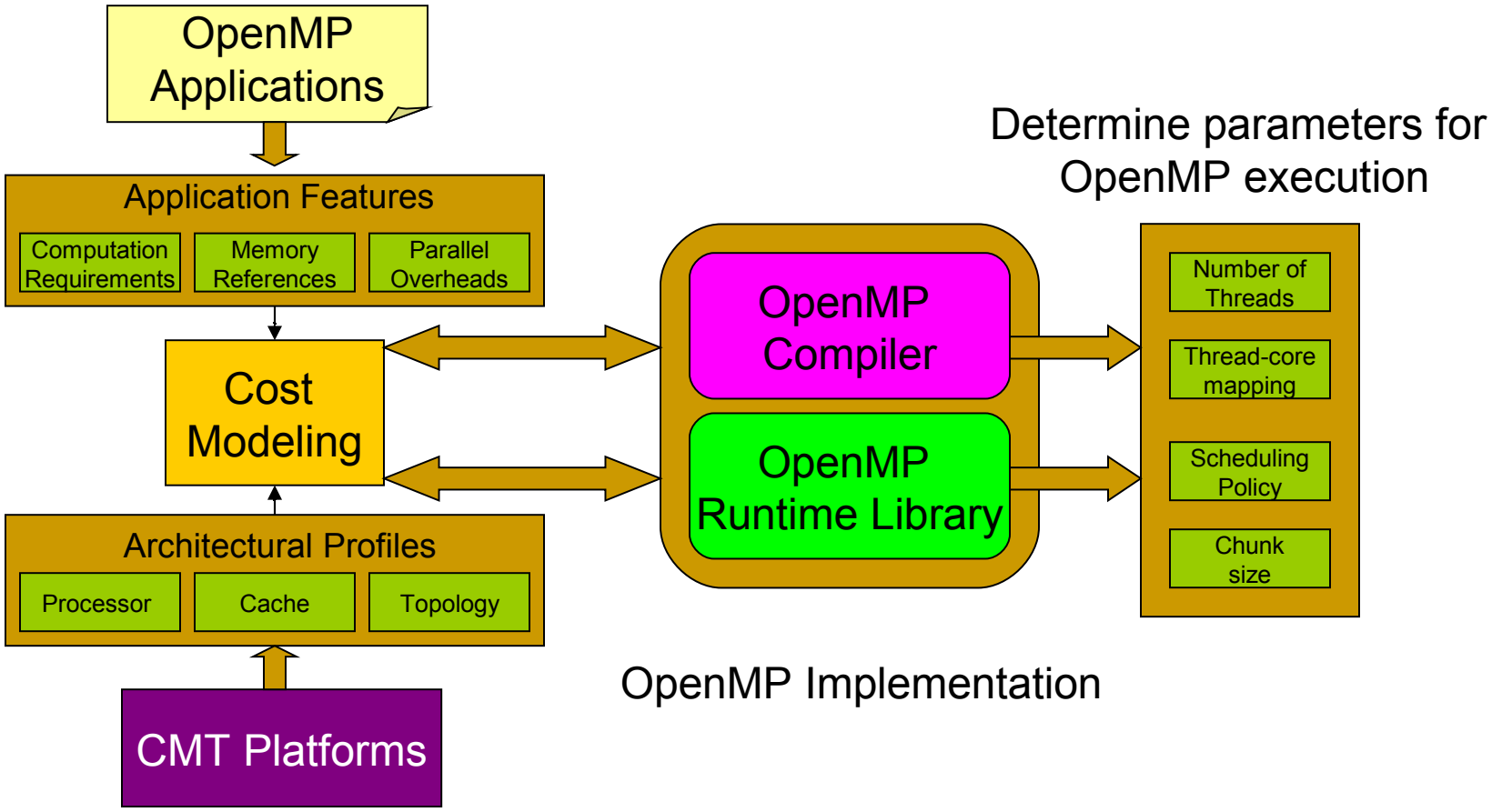
# Example Usage of Cost Modeling

```
DO K2 = 1, M, B
  DO J2 = 1, M, B
    DO I = 1, M
      DO K1 = K2, MIN(K2+B-1,M)
        DO J1 = J2, MIN(J2+B-1,M)
          Z(J1,I) = Z(J1,I) + X(K1,I) * Y(J1,K1)
```




- Case 1: What is the optimal tiling size for a loop tiling transformation?
  - Cache size,
  - Miss penalties,
  - Loop overhead,
  - …
- Case 2: What is the maximum number of threads for parallel execution without performance degradation?
  - Parallel overhead
  - Ratio of parallelizable_work/total_work
  - System capacities
  - …

# Potential Use of OpenMP Cost Modeling

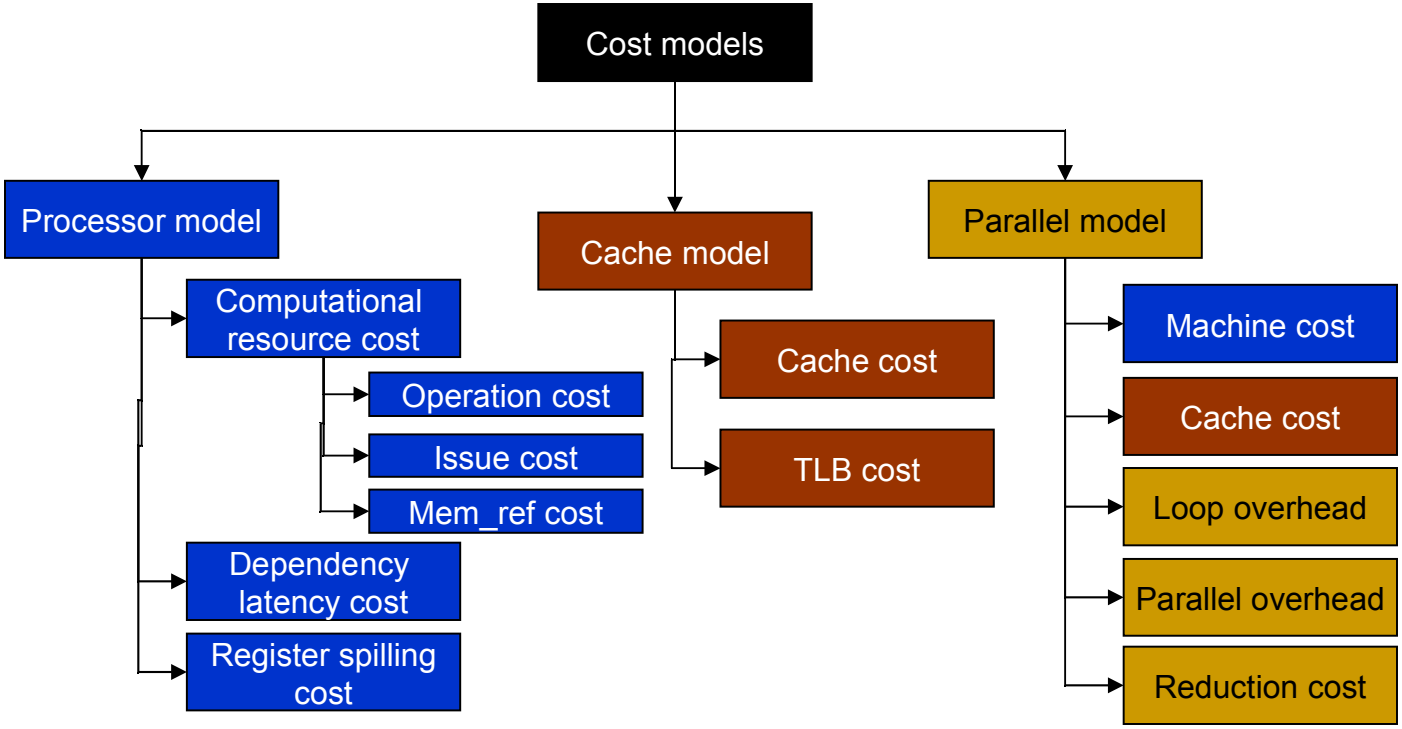


# Modeling OpenMP

- Previous models
  - T_parallel_region = T_fork + T_worksharing + T_join
  - T_worksharing = T_sequential / N_threads
- Our model aims to consider much more:
  - Multiple worksharing, synchronization portions in a parallel region
  - Scheduling policy
  - Chunk size
  - Load imbalance
  - Cache impact for multiple threads on multiple processors
  - …

# Implementation in OpenUH

- Extend sequential model
  - Load imbalance
  - Scheduler
  - Currently number of threads provided in environment variable



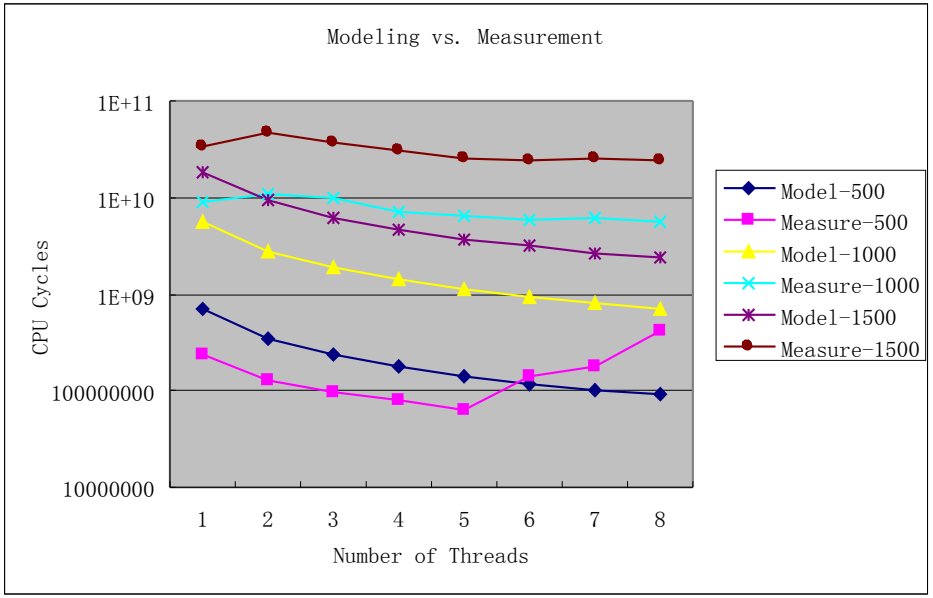
# An Experiment

- Machine: Cobalt in NCSA (National Center for Supercomputing Applications)
  - 32-processor SGI Altix 3700
  - 1.5 GHz Itanium 2 with 6 M L3 Cache
  - 256 G memory
- Benchmark: OpenMP version of a classic matrix-matrix multiplication (MMM) code
  - i, k, j order
  - 3 different double floating point matrix sizes:  500, 1000, 1500
  - OpenUH compiler: -O3, -mp
- Cycle measuring tools:
  - pfmon, perfsuite

```
#pragma  omp  parallel  for   private ( i , j , k )
f o r ( i = 0 ; i < N; i ++)
  f o r ( k = 0 ; k < K; k++)
    f o r ( j = 0 ; j < M; j ++)
      c [ i ] [ j ]= c [ i ] [ j ]+ a [ i ] [ k ]*b [ k ] [ j ] ;
```

# Results

- Efficiency = Modeling_Time / Compilation_Time x100% = 0.079s/6.33s =1.25%




Model captures trends, but not scale

- Measured data have irregular fluctuation, especially for smaller dataset with larger number of threads
  - $10^8$ cycles@1.5GHz <0.1 second, relatively big system level noise from thread management
- Overestimation for 500x500 array from 1 to 5 threads, underestimation for all the rest:
  - optimistic assumption for resource utilization
  - more threads, more underestimation

Lack of contention models for cache, memory and bus

# Cost Model

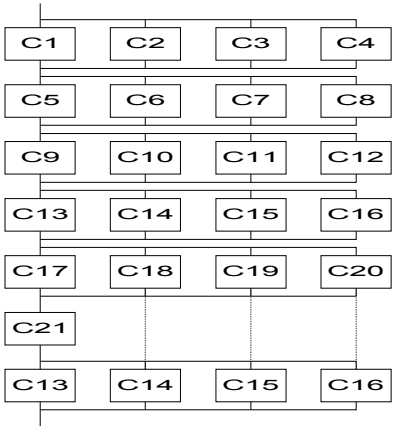
- Detailed cost model could be used to recompile program regions that perform poorly
  - Possibly with focus on improving specific aspect of code
- Current models in OpenUH are inaccurate
  - Most often they accurately predict trends
- Fail to account for resource contention
- This will be critical for modeling multicore platforms

- What level of accuracy should we be going for?

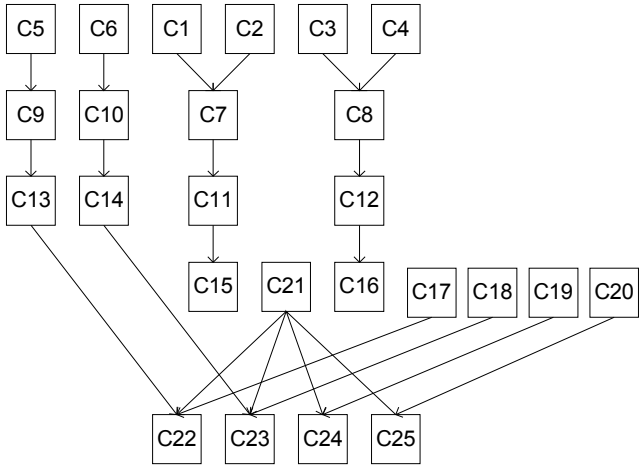
# A Longer Term View

```
!$OMP PARALLEL
!$OMP DO
       do i=1,imt
          RHOKX(imt,i) = 0.0
        enddo
!$OMP ENDDO
!$OMP DO
       do i=1, imt
          do j=1, jmt
             if (k .le. KMU(j,i)) then
                RHOKX(j,i) = DXUR(j,i)*p5*RHOKX(j,i)
             endif
          enddo
       enddo
!$OMP ENDDO
!$OMP DO
       do i=1, imt
          do j=1, jmt
             if (k > KMU(j,i)) then
                RHOKX(j,i) = 0.0
             endif
          enddo
     enddo
!$OMP ENDDO
       if (k == 1) then
!$OMP DO
       do i=1, imt
          do j=1, jmt
             RHOKMX(j,i) = RHOKX(j,i)
          enddo
       enddo
!$OMP ENDDO
!$OMP DO
       do i=1, imt
          do j=1, jmt
             SUMX(j,i) = 0.0
          enddo
       enddo
!$OMP ENDDO
     endif
!$OMP SINGLE
       factor = dzw(kth-1)*grav*p5
!$OMP END SINGLE
!$OMP DO
       do i=1, imt
          do j=1, jmt
             SUMX(j,i) = SUMX(j,i) + factor * &
                         (RHOKX(j,i) + RHOKMX(j,i))
          enddo
       enddo
```

Part of computation of gradient of hydrostatic pressure in POP code




Runtime execution model for (c stands for chunk)




Dataflow execution model associated with translated code

# Summary

- Language features needed to enhance expressivity, scalability of OpenMP
  - For large DSMs, and for multicore platforms
  - OpenUH permits realistic experimentation with new features
- There is much to explore in implementation of OpenMP on multicore
  - Alternative translation styles
  - Ability to model new platforms statically
  - Dynamic adaptation
- Open64 technology a good basis for such efforts

# Questions?